Autonomous Dynamically Self-organizing and Self-healing Distributed Hardware Architecture – the eDNA Concept

Michael Reibel Boesen[1], Jan Madsen[1] and Didier Keymeulen[2].

[1] Technical University of Denmark
Richard Petersens Plads, Bygning 322
Kgs. Lyngby, Denmark 2800
+45 45752505
{mrb,jan@imm.dtu.dk}

[2] Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
818-354-4280
{didier.keymeulen@jpl.nasa.gov}

Abstract—This paper presents the current state of the autonomous dynamically self-organizing and self-healing electronic DNA (eDNA) hardware architecture (patent pending). In its current prototype state, the eDNA architecture is capable of responding to multiple injected faults by autonomously reconfiguring itself to accommodate the fault and keep the application running. This paper will also disclose advanced features currently available in the simulation model only. These features are future work and will soon be implemented in hardware. Finally we will describe step-by-step how an application is implemented on the eDNA architecture.

TABLE OF CONTENTS

1. INTRODUCTION

In the age of ubiquitous computing all parts of the industry is in need of highly robust hardware platforms. Not only due to the fact that embedded systems are given increasingly often life-saving, life-depending roles, such as autonomous subway systems, airplanes, cars, hospital equipment etc. An unprotected hardware fault in either of these will have dire consequences – and consequently hardware faults in such systems are always protected by huge amounts of redundancy. But even the state-of-the-art hardware fault prevention technique – Triple Modular Redundancy (TMR) has its limits. A fault in the voter circuits or a permanent fault in one of the copies will eliminate the TMR's ability to identify the correct value, while a repair of the faulty module will allow it to reconstruct the TMR. The capability of a hardware platform to autonomously be able to repair itself becomes particularly important in space, where a repair mission will be either a great risk, impossible or very expensive or all of the above.

In the last decade several biologically inspired reconfigurable self-healing hardware platforms have been proposed [3,4,5,6] all of these suffer from problematic scaling issues due in particular to a too low level of logical granularity. Consequently, (to the best of our knowledge) neither of these has ever been applied to a real world application.

Other approaches, such as roving STARS [8] uses a centralized approach, where a centralized processing unit is responsible for performing the fault tolerance mechanism. Clearly, approaches using a centralized unit have single-point-of-failure properties, which in a high reliability environment would be unacceptable.

The eDNA architecture [1,2] is aimed for an ASIC implementation and will consequently be an entirely new type of fault-tolerant coarse grained FPGA due to the increased level of logical granularity, when compared to other approaches. The increased level of logical granularity makes the cost of the self-healing feature bearable [2].

The following section provides an overview of the fundamental concepts of the eDNA system, section 3 describes the details of the eDNA concept. Section 4 illuminates how the eDNA architecture is capable of Self-programming. Section 5 illuminates how the eDNA architecture is capable of Self-healing. Section 6 describes the implementation of the prototype and its current limitations. Section 7 illustrates how to implement an application on the eDNA prototype. Section 8 describes advanced features such as the fault-detection protocol and dynamical application scaling not yet implemented in the eDNA prototype but implemented in the simulation model of the final eDNA architecture. Finally, section 9 presents our conclusion.

2. eDNA: Fundamental concepts

eDNA is the name of the entire package described in this paper – consequently, we have two fundamental terms; the eDNA *architecture* and the eDNA *program*.

The eDNA architecture consists of a distributed array of multiple homogenous processing units called *electronic cells* (eCells). The job of the eCells is to implement the eDNA program, which is specified by the programmer. The eDNA program is translated into a binary version of the eDNA, which is then fed to all eCells which all store it a RAM block. Each eCell implement a part of the eDNA program. The specific part, which an eCell implement is, called the gene of this particular eCell.

Each eCell contain a microprocessor and a 32 bit ALU which is configured by the microprocessor to perform a certain function described by the gene. The program run by the microprocessor is termed the *ribosomal DNA* (referring to the intracellular organelle in biological cells, responsible for synthesizing proteins and consequently functionality of the cells). The ribosomal DNA is a program written for the eCell microprocessor, which performs *self-programming* and *self-healing* of the eDNA architecture. All eCells contain a homogenous copy of this program.

Observe that no centralized processing unit is present. The eCells completes the self-programming and self-healing completely autonomous.

The eCells communicate with each other through a Network-on-Chip (NoC) 2D-mesh-8 architecture, where each eCell communicate with at most 8 adjacent neighbors depending on position. The NoC completes package transfers between eCells using a fault-tolerant data-transfer protocol, which can route around dead links.

Figure 1 shows an overview of the entire eDNA package.

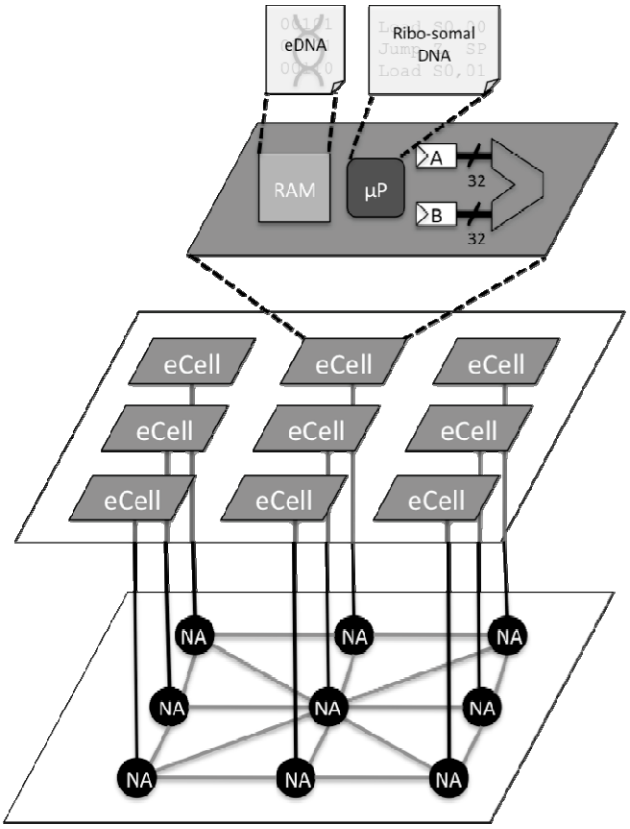

Figure 1 An overview of the eDNA concept

3. eDNA explained

The two fundamental concepts of the eDNA architecture is the electronic DNA (eDNA) and the programming model used to map it onto the eCells of the eDNA architecture.

The electronic DNA (eDNA): Concept

The eDNA is a program written in a programming language known as the eDNA language [1,2]. The BNF notation of the language is shown in Figure 2. As can be seen the eDNA language contains assignments, control structures such as if-then-else and loops, as well as explicit control for parallelism (the `<parallel>` syntax). With the `<parallel>` syntax the programmer can mark which parts of the code should be executed in parallel.

```
dna        ::= <statement>* |  <parallel>*
statement  ::= <assignment> | <while> | <if> |  return <var/c> | <parallel>
parallel   ::= parallel <statement>* endparallel
assignment ::= <var/c> = <exp>
while      ::= while <bexp> do <statement>* endwhile
if         ::= if <bexp> then <statement>* else <statement>* endif
exp        ::= <var/c> [<op> <exp>]*
bexp       ::= <var/c> [<bop> <bexp>]*
op         ::= AND | OR  | + | - | ...
bop        ::= AND | OR | < | <= | == | != | ...
var/c      ::= Letters{A-Z}* | <const> | RAM <var/c>
const      ::= 0<const>* | 1<const>*
```

Figure 2 The eDNA language

There are two parts to the programming model: (1) Synthesis and (2) mapping. The synthesis is inspired by Ian Page [7]. Ian Page proposed a way of synthesizing software code directly in hardware. The idea was to replace the individual parts of a programming language with hardware blocks. We have adapted these blocks to fit the eDNA architecture. The main four parts of the eDNA language (assignments, if-then-else, while-loop and parallel) can be seen on Figure 3(a)-(d) as well as their hardware block counterparts. There are two components to each block; (1) synchronization and (2) logic. When a program is executed we want the order of instructions to stay the same as in the program. This is synchronized using the *start/finish signaling*. Each of the four blocks in Figure 3 features a start/finish signal. Whenever the data-flow of the program reaches a particular block, the start signal is kept high. Whenever a block is finished executing, a finish signal will be sent out, which will become the start signal of the next block, hereby activating it. Finally, the *logic* is the logical expression of the program statement. For instance in the if-then-else block (Figure 3(b)) the part denoted by *IF-G BOOL* is the logical expression of the if-then-else sentence, which consists of the Boolean operation related to the sentence plus some logic to control whether to execute Statement S1 or Statement S2 – i.e. if the if condition is true S1 will receive a high start signal and S2 will receive a low start signal.

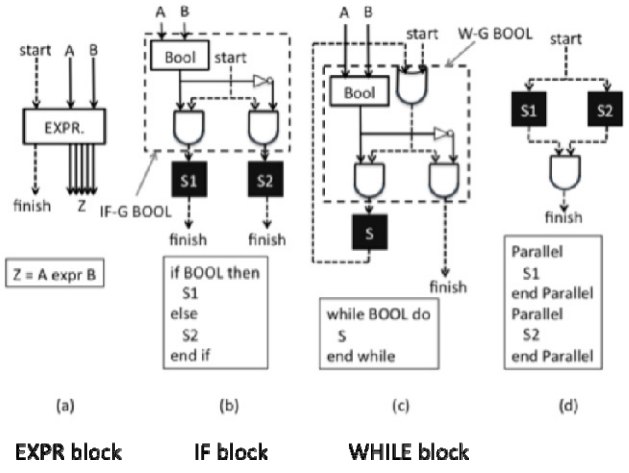

Figure 3 Synthesis of eDNA program code

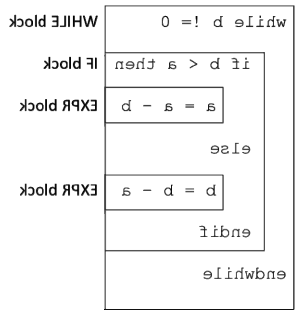

Figure 4 Program divided into the blocks of Figure 5

Figure 4 shows an example of an eDNA program, showing how to apply the blocks in a modular way to it. On the top level we have the while block Figure 3(c), inside this we have an if-block Figure 3(b) and finally, inside this one we have two EXPR blocks Figure 3(a).

Formal model of the eDNA program

By applying these blocks to the eDNA program code we can derive a task-graph $\Gamma=<V,E>$, where V is a set of vertices and E is a set of edges. The taskgraph divides the program into several smaller eDNA-tasks.

An eDNA-task $\tau_d \in V$ is physically represented on the eDNA architecture as Figure 3(a), the IF-G BOOL part of Figure 3(b) or the W-G BOOL part of Figure 3(c). Where d is defined as the depth of τ_d in the task-graph found using Breadth-First-Search – consequently, the source node in the task-graph is τ_0 and the sink τ_{N-1} where N is the number of eDNA-tasks in Γ. Furthermore, we will make the restriction that all tasks in Γ will have a unique index – even if two tasks are parallel will they be given two different indices.

An eDNA edge $\varepsilon_{i=>j}\{start,data\} \in E$ represent communication between τ_i and τ_j. Furthermore, an eDNA edge has a type associated with it, which can either be *start* or *data*. A *start* type edge directly represents the start/finish signaling of Figure 3. A *data* type edge represents the Z output from Figure 3(a). This implies that the source node of this edge must be of the type of Figure 3(a), since only the EXPR block can output data.

An example eDNA task graph can be seen in Figure 5. The task graph implement the eDNA code seen in Figure 6. As an example of the edge-definition take the data edge going from τ_3 to τ_0. Using the definition, this edge would be known as $\varepsilon_{3=>0}$ *{data}*.

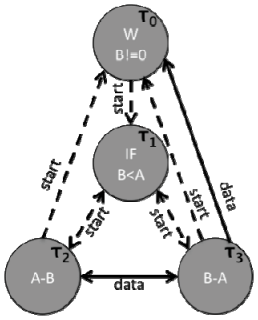

Figure 5 Task-graph representation of an eDNA program.

```
while b != 0 do
   if b < a then
      a = a - b
   else
      b = b - a
   endif
endwhile
```

Figure 6 An example of eDNA program code

Programming model: eCell programming

We now know how to get from a program description to hardware. Next part is to realize this hardware on the eDNA architecture. For this purpose we will introduce the concept of eCell types. An eCell type is directly related to the task τ_d it implements, consequently eCells can be either an EXPR-cell, IF-cell or a WHILE-cell – corresponding to Figure 3(a)-(c), respectively. This means that an eCell, which is implementing an EXPR-type task, is an EXPR-type eCell. An EXPR-type eCell will have to contain 3 registers (Z, A and B) and an ALU, which can perform the expression applied to A and B. An IF-cell and WHILE-cell contains the logic needed to evaluate the boolean condition in order to decide where to send the start signal. Note, that the eCells are homogenous, meaning that all eCells have the potential to become either one of the types. This means that each eCell basically contains an ALU, which can be reconfigured according to the task this eCell is required to perform, i.e. whether to be an EXPR cell, an IF-cell or a WHILE-cell (Figure 7). This leads to a much higher level of logical granularity than previous self-healing architectures [3,4,5,6,8]. The eDNA architecture is closer to a reconfigurable datapath array (rDPA) than an FPGA in terms of logical granularity.

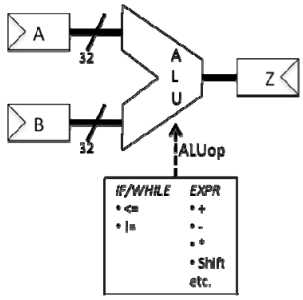

Figure 7 eCell Type Selector

Programming model: Mapping

In order to explain how to map eCell types to eCells we first have to elaborate on the eDNA architecture. A schematic of the eDNA architecture is shown in Figure 8. The figure shows the key parts of the eDNA architecture. Each square represents an eCell. Note that the operator in the middle of the square defines the eCell type and consequently, represents the task τ_d that the eCell implements. Observe that some of the eCells does not have a type; these eCells spare-eCells. The spare-eCells contain exactly the same hardware as the eCell with a type, but no task has been mapped to it.

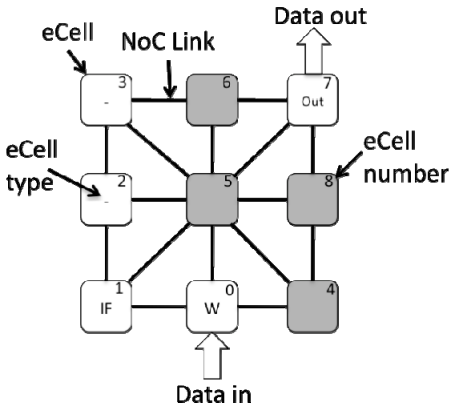

Figure 8 A 3x3 eDNA architecture

Mapping eCell tasks to the eDNA architecture is a matter of creating the mapping of a task τ_i to an eCell C_j, such that $M(\tau_i) \rightarrow C_j$ and where $i=j$, where i is the *eCell number* of an eCell as seen on Figure 8. The eCell number is an integer from 0 to K-1 (where K is the total number of eCells), and each eCell has an unique number. A task τ_d, which is mapped to an eCell C_d, is defined as the gene of the eCell C_d's.

Observe that this means that there's a 1:1 relationship between the eCells and the example task graph of Figure 5, i.e. each eCell implement one task.

eCell numbers are distributed to the eCells when the eDNA architecture is initialized and can also be changed

dynamically during the execution of an application. This means that the positioning of the eCell numbers on the eDNA architecture defines the position of a particular task of the application. Obviously, the position of the tasks on the array impacts performance of the application, because the transfer of data in the NoC will use additional time pr. link, which has to be traversed. Therefore, we have developed a metaheuristic algorithm based on Tabu Search, which offline optimizes the mapping of identifiers to eCell positions in the network. This is an optimization issue and consequently not the scope of this paper.

Programming model: eDNA Program Representation

In order for an eCell to implement a task τ_d the following information about the task are needed:

1. The type of τ_d (Figure 3).
2. The type of all edges $\varepsilon_{d=>j}\{start,data\} \in E$ (the edges for which τ_d is the source) (Figure 5)
3. All eCell numbers *j* for all edges $\varepsilon_{d=>j}\{start,data\}$ (all eCell numbers – and consequently task indices - that the outgoing edges of the task τ_d points to) (Figure 5).
4. For all tasks τ_i, where $i=0..., N$ the mapping $M(C_i) \rightarrow (X_i,Y_i)$, where X_i and Y_i are the coordinates of C_i in a 2D grid.

(1) is the ALUop signal from Figure 7. (2) is a variable telling the eCell whether the type is *start* or *data* (1). (3) is the relative address i.e. eCell number to communicate with. (4) is a mapping of eCell numbers to absolute locations (X,Y in 2D grid) in the 2D-mesh NoC, which utilizes an adaptive XY-routing algorithm capable of routing packages around dead links and faulty eCells. This mapping is in its essence a routing table. Consequently, the combination of (1)-(3) is the information stored in a gene of an eCell.

An example of a routing table and genes are shown in Figure 9. The example in Figure 9 shows the gene related to the `b=b-a` in the eDNA program segment seen of Figure 6. On top we have the routing table. The left column contains the eCell number, which is then related to the physical address in the network in the right column. This physical mapping of eCell numbers to coordinates can also be seen in Figure 8. The lower table contains the gene, which consists of multiple *gene-instructions*. As seen the three pieces of information mentioned earlier are present (eCell type, edge type, communication target) plus one additional piece of information, which is a program counter (PC). The PC tells the eCell the address of the next gene-instruction to interpret. A gene can consequently, be compared to an instruction in a microprocessor. Note that this gene can be directly derived from the task graph of Figure 5. Also note that when a particular gene for an eCell ends, the eDNA contains an empty gene-instruction.

The eCell executes the genes in the following way:

1. Start at PC=00
2. Program and execute the ALUop according to the eCell type
3. Attach a package header which is equal to the edge type
4. Target the package at the relative address under "Target".
5. Translate relative address to absolute address by using the routing table.
6. Send package.
7. Increment program counter according to PC field
8. Repeat from 1 until an empty gene is reached.

Identifier	Address
00	(2,1)
01	(1,1)
02	(1,2)
03	(1,3)

PC	eCell type	Edge type	Target
01	EXPR(B=B-A)	Data	02
02	EXPR(B=B-A)	Data	01
03	EXPR(B=B-A)	Data	00
00	EXPR(B=B-A)	Start	00

Empty gene

Figure 9 eDNA representation example: Routing table and genes

To handle this, we need more logic than described by Figure 7. The new eCell can be seen in Figure 10. The ALU from Figure 7 is still present, but we have added 2 RAM blocks (eDNA RAM and Gene RAM) and an eCell processor. The eDNA RAM contains the entire binary eDNA program and the Gene RAM contains the gene of this particular eCell. The eCell processor controls the self-programming and self-healing.

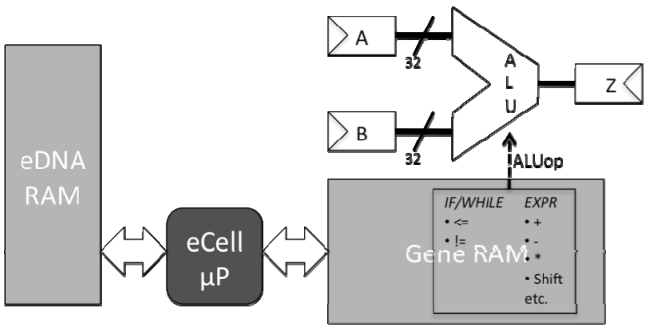

Figure 10 eCell with processor and RAM

4. SELF-PROGRAMMING

Upon initialization the eDNA program in its binary form is distributed to all eCells together with the routing table (Figure 9 in binary encoding) and is stored in the eDNA RAM. All eCells consequently contains the eDNA program code and routing table.

With the eDNA program in the eDNA RAM we can now start the self-programming which for each eCell C_d consists of two steps:

1. Localization of task/gene τ_d.
2. Copying of the task/gene τ_d from eDNA RAM to the Gene RAM

Both steps are simple; the localization is simply to count the number of empty gene-instructions in the eDNA. Whenever this count is equal to C_d, the eCell has located its gene. The copying of the code from the eDNA RAM to the gene RAM block is trivial.

5. SELF-HEALING

The autonomous self-healing mechanism consists of 5 phases:

1. Fault-detection
2. Spare-eCell localization
3. Healing of the faulty eCell
4. Link repair
5. Data maintenance

Fault-detection

Whenever a fault occurs, we have to detect it. We have devised a fault-detection mechanism, which utilizes the homogenous nature of the eCells to detect faults in a way similar to how triple modular redundancy (TMR) works.

Each eCell has a gene, which is activated and executed upon reception of a start signal. This gene we denote the *primary gene*. Similarly, we define a secondary (set of) genes, denoted *secondary genes* and a *secondary start* signal that will activate and execute the secondary genes. The secondary genes for an eCell C_d are defined as:

$$\forall \tau_i \in V,\ \exists \varepsilon_{i=>d}\{start\} \in E$$

i.e. all tasks τ_i in V for which there exists and start-type edge, whose destination is C_d. The application of this definition to the task-graph of Figure 5 is seen in Figure 11. The idea is that this eCell will test the output of the eCell located one step behind in the execution of the program. Note that in the case of a branch (IF-cell), this might be two genes. Refer to the WHILE eCell of Figure 5 for an example. The WHILE eCell's secondary genes are the gene of the A-B eCell and the B-A eCell, because both of these eCells send the start package to the WHILE eCell.

Just as primary genes has a start signal secondary genes has too. The secondary start signal, which activates the secondary gene(s) are sent by the C_i to the C_d for which the following holds:

$$\exists(\varepsilon_{i=>i}\{start\},\ \varepsilon_{i=>d}\{start\})$$

This means that it is sent by the eCell located two execution steps behind C_d. The reason for this is that we want to make the impact of the fault detection protocol as little as possible on the performance of the application and in this way we can do the fault detection in semi-parallel to the real execution. In the case of the WHILE eCell of Figure 5 the secondary start signal is sent by the IF eCell. Upon reception of a secondary start signal the eCell will execute the secondary gene(s). A secondary gene *only* consists of an ALUop gene describing what the eCell this eCell is supposed to test is doing in its ALU, i.e. only information about the eCell type is needed. Note that this means that we will need 4 more registers (2x2 in case of an IF cell) in order to be able switch between executing primary genes or secondary genes. Figure 11 shows Figure 5 augmented with secondary genes. Observe that in order to provide the data for the secondary genes data edges might have to be added. In the case of Figure 11, we have to add an edge going from τ_2 to τ_0, because τ_0 didn't need A before.

Whenever a primary start signal is sent from C_i it also sends the result of its primary gene with this package. The receiving eCell C_k has now already calculated its own result of this calculation as its secondary genes, so it compares

this result to the result arriving in the primary start package. If they're equal no fault is assumed if they're not equal we have a fault – but as of yet we don't know in which eCell.

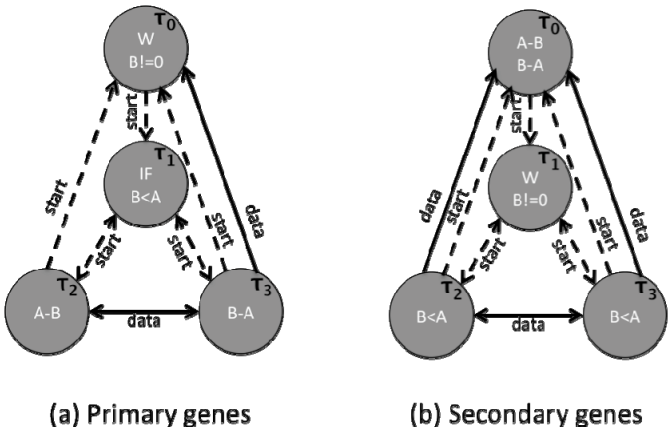

Figure 11 (a) The primary genes of Figure 5 augmented with secondary genes in (b)

Therefore, in the case of a fault C_k sends a "test-package" to a the nearest spare-eCell C_p. This package contains the eCell number C_i, the A and B registers, plus the two results. C_p now examines the eCell number (C_i) contained in the package and self-organizes to implement the same task τ_i. It then computes the result based on the value of the registers it received and decides which C_i or C_k that is faulty. Hereby we have detected a fault. The eCell C_p, which performs the final test is in the following denoted the *fault-detector*.

This is similar to TMR because in TMR we also have one copy, which implicitly decides which set of copies is right and which is wrong. Two differences between the eDNA fault detection protocol and TMR exists: (1) eDNA architecture complete the fault-detection sequentially where it is done in parallel in TMR, and (2) the fault-detector is different from fault to fault and consequently the voter who decides which eCell is faulty is different from time to time – giving us a higher probability of avoiding voter caused errors.

Spare-eCell localization

Observe that because the fault detection is performed each time a primary start signal is received (i.e. when an edge $\varepsilon_{d=>j}\{start\}$ is traversed) only one fault is detected at any one time even though multiple faults may be present – this is ok since the fault will not have any impact until the faulty eCell is executed. This simplifies the way the eCells determines where to move the functionality of the faulty eCell, because they don't need to take into account that other eCells might be moving functionality around at the same time.

Implicitly included in the eDNA is a list of spare eCells. A spare eCell is an eCell C_s for which it holds that

$$0 \le i < K.\ \forall \tau_i.\ \neg\exists M(\tau_i) \to C_s$$

i.e. if an eCell C_s hasn't got a task mapped to it, consequently the eCells which are not in the routing table of the eDNA. Locating the best spare-eCell is done by computing the Manhattan distance to all spare-eCells and selecting the closest. Note in case of many faults we cannot guarantee a globally optimal selection of spare-eCell with this protocol, however, this is in general and impossible problem to solve, since we have no way of knowing when and where faults will occur. This information will be needed in order to ensure a globally optimal selection of spare-eCell.

Healing of the faulty eCell

When we have selected a spare-eCell C_s, all that is needed to replace the functionality of the faulty eCell C_f at C_s, is to change the mapping M_f of C_f to C_s, consequently:

$$M(\tau_f) \to C_f \text{ a } M(\tau_f) \to C_s$$

All that is needed to change the mapping is to write a different coordinate in the routing table for C_f, which was faulty. Eg. if eCell 00 of Figure 9 was faulty we would simply need to rename the entry for eCell 00 to the coordinates of C_s. The new routing table could for instance become as seen in Figure 12.

In order to do that the fault-detector will need to broadcast a *heal-package* to all eCells simply saying to replace the coordinates at eCell number 00 with (2,2). This replacing is done in the eDNA RAM, so consequently all eCells will be reinitialized after that. This will now cause the eCells who used to communicate with eCell (2,1) to communicate with eCell (2,2) and it will cause eCell (2,2) to know that it is a part of the application. Consequently eCell (2,2) will now copy the genes of the faulty eCell (2,1) from its own eDNA RAM to its Gene RAM. Hereby we have restored the functionality of the C_f using nothing but the information stored in the eDNA. The resulting physical remapping can be seen in Figure 13.

Identifier	Address
00	(2,1)
01	(1,1)
02	(1,2)
03	(1,3)

Autonomous remap

Identifier	Address
00	(2,2)
01	(2,1)
02	(1,2)
03	(1,3)

Re-mapped from (2,1) to (2,2)

Figure 12 Self-healing example

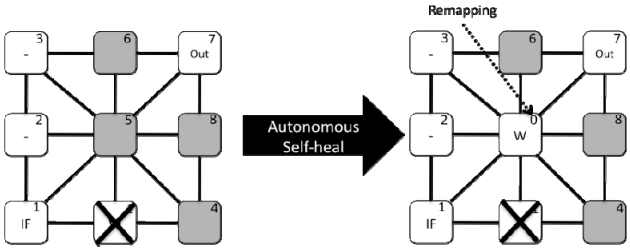

Figure 13 Result of the autonomous remapping and self-heal

Connection repair

Observe that edges, which used to point at C_f now will point C_s due to the reinitialization.

Data maintenance

In order for eDNA to continue without outputting faulty data C_s will need to restore the same data (i.e. register A and B of Figure 10). Static data is no problem, since the eDNA also contains any initialized values of these registers. So when C_s is reinitializing to repair C_f it will automatically gain the initialized values of the A and B registers. However, dynamic data is harder, since the value of it depends on the time at which the fault occurred. Fortunately, the fault detection protocol automatically keeps track of this data using the secondary genes – due to the new edges (example Figure 11) we have to add. So when the fault detector has sent the heal-package it will send the contents of register A and B. In this way data is recovered.

6. PROTOTYPE

We have developed a prototype of the architecture described in the preceding sections. So far we have implemented everything except step 1 and 2 of the self-healing mechanism. Thus we rely on fault-injection to test it.

The eCell

The hardware architecture of the eCell implemented in our prototype can be seen in Figure 14.

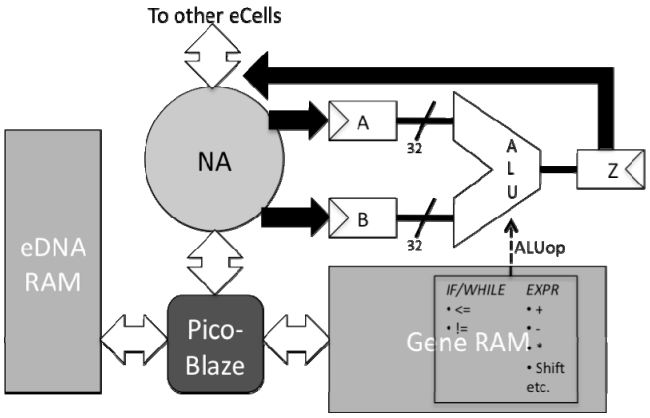

Figure 14 Prototype eCell

This architecture is in contrast to classical NoC's that consists of separated routers and NA's. The combination of router and NA is meaningful for this setup due to the chosen level of granularity of the eCells, the processing time where the CPU blocks the NA is short and the package size is small. Therefore eCells got extended with a store-and-forward (SAF) routing functionality. This leads to a simplified homogeneous structure of the system, reduces the number of hops and eases the failure detection, which will be implemented in the final design.

The NA consists of a pair of peripheral switches, a number of registers that are capable of storing a single package and a state machine that is capable of handling signaling, package transfers, routing algorithm, triggering the CPU interrupt and controling the register accesses. A schematic of the NA is seen in Figure 15.

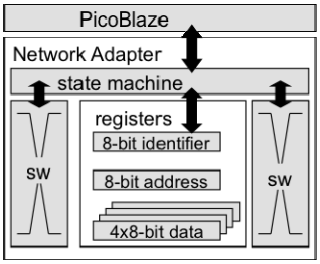

Figure 15 Network adapter

In order to implement the self-programming and self-healing algorithm in the most flexible way and to explore different networks and applications we decided to run a soft core CPU connected to the NA in each eCell. In the final version of the eDNA platform this CPU will be substituted by dedicated logic in order to reduce chip area, complexity and to increase speed. Due to the small amount of resources required and its flexibility the PicoBlaze was chosen. The Xilinx PicoBlaze [9,10] is based on a 8 bit RISC architecture that is optimized in size for Virtex and Spartan series of FPGAs. It is provided as a synthesizable source-

level VHDL file which is easy to extend with additional ports to fit the desired application. There are assembler and C compiler available. It was possible to implement a test setup consisting of a 7x7 array of eCells (PicoBlaze and NA) in a mesh-8 6x8 bit (6 parallel 8 bit registers) package architecture on a Digilent XUPV2P board. This setup also includes dedicated input and an output eCells that connect to a serial interface, allowing the user to input data and commands to the system and read the results on a terminal.

The first 8 bit of the package are used as package identifier, which describes the content of the data (e.g start/finish signaling, variable read/write...). The 8 address bits are split up in 4 bit X/Y-coordinates, the address space is therefore limited to 225 cells (0 is no valid address) which is meaningful for test implementations on FPGAs. The remaining 4x8 bit are used to carry data.

Network Topology and Routing

The distributed approach used in the eDNA platform creates a significant over-head by transmitting data packages among the eCells. In addition to the data exchange, the start/finish signaling is implemented purely package-based and will generate additional traffic in the NoC. It is therefore very important that the prospective network is able to forward packages in a simple and fast manner since the performance of the whole system greatly depends on the network properties.

The NoC is implemented as a 2D mesh-8 topology, which means that each eCell is connected to its N, NW, W, SW, S, SE, E and NE neighbor when applicable (Figure 1).

When a NA of an eCell receives a package, it checks whether the destination address of the package is reached. If not, then the NA determines the direction of the destination eCell and sets the output switch to the corresponding position. Only if the package destination address is reached, the CPU is interrupted to perform the eDNA functionality. When the CPU has generated a new package, the NA checks whether the destination is available by handshaking. This ensures package rerouting in case of network faults.

Before transmitting, the NA makes sure that the next eCell on the direct way to the destination eCell is ready. This is done by handshaking and also includes a dedicated signal, which reports whether the corresponding eCell is alive or not. In case the next eCell is busy, the package is sent to one of the neighboring eCells. This mechanism ensures that dead or busy eCells on the way to the destination can be sidestepped. In the eDNA prototype platform, SAF routers are used and a datagram is equal to a flit. This flow control digit is defined as the smallest unit of flow control. Due to these design decisions and the fact that only one package per `parallel` statement is routed in the network at the time, deadlocks and livelocks can be avoided.

Implementation in embedded system

In other work, for the purpose of studying a real world application we ported the eDNA prototype from the Xilinx 5 FPGA to a National Instruments CompactRIO embedded system, consisting of a 800MHz PowerPC running VxWorks, a Xilinx 5 FPGA and analog I/O modules. The eDNA prototype was implemented on the Xilinx 5 FPGA of the CompactRIO and used to control and do data processing for a Fourier Transform Spectrometer. Results are reported here [11].

7. USE-CASE EXAMPLE

In this example we will implement the Discrete Cosine Transform (DCT) on the eDNA architecture.

The DCT is a widely used mathematical transform in signal processing. It expresses a finite sequence of data points as a sum of cosine functions, which oscillate at different frequencies. It is used in wide array of applications, such as image compression and spectroscopy. In this example we will implement the one dimensional DCT on the eDNA architecture.

The 1-D DCT is defined by the following equation:

$$Y_k = \sqrt{\frac{2}{N}} \alpha_k \sum_{n=0}^{N-1} x_n \cdot \cos \frac{(2n+1)k\pi}{2N} \qquad (1)$$

where Y_k is the *kth* element of the DCT of the data set $X=\{X_0,\ldots,X_{N-1}\}$ and

$$\alpha_k = \begin{cases} \frac{1}{\sqrt{2}} & k = 0 \\ 1 & k = 1,..,N-1 \end{cases} \qquad (2)$$

The following will explain the simple implementation procedure to follow to implement an application on the eDNA architecture.

Application to eDNA program

Equation (1) and (2) can be implemented by the eDNA program Figure 16. Note that since each eCell do one binary arithmetic operation we have to split the formula into several eCells.

```
while k <= N1 do
    if (i != 0) then
        α = α1 + 0
    else
        α = α0 + 0
    endif
    const = C * α
    while (n <= N1)
        p1 = 2 + 0
        p2 = p1 * n
        p3 = p2 + 1
        p4 = p3 + k
        p5 = p4 * pi
        p6 = p5 >> LN
        p7 = cos(p6)
        p8 = p7 * Xn
        p9 = p9 + p8
        n = n + 1
    endwhile
    n = 0 + 0
    k = k + 1
    Yk = const * p9
endwhile
return out
```

$$\alpha_k = \begin{cases} \frac{1}{\sqrt{2}} & k = 0 \\ 1 & k = 1..,N-1 \end{cases}$$

$$\sqrt{\frac{2}{N}}\alpha_k$$

$$p5 - (2n+1)k\pi$$

$$p6 = \frac{p5}{2N} = (p5)\,\mathrm{SRL}\,(-\log 2(2N))$$

$$\sum_{n=0}^{N-1} x_n \cdot \cos\frac{(2n+1)k\pi}{2N}$$

$$Y_k = \sqrt{\frac{2}{N}}\alpha_k \sum_{n=0}^{N-1} x_n \cdot \cos\frac{(2n+1)k\pi}{2N}$$

Figure 16 DCT eDNA

Compilation of eDNA program to eDNA for the eCells

This eDNA program is typed in the editor of our eDNA SW Toolkit. The only other information needed is where the user want the different genes mapped, so the user need to supply the routing table OR alternatively the user could use our Tabu Search based algorithm (shortly mentioned in Section 3) to find a good solution to the mapping problem.

The toolkit will then return the encoded eDNA (of the same type as Figure 9) for the eCells, a task graph and a software model of the eDNA, which can be used to perform simulations on the software model of the eDNA architecture implemented in the toolkit. Figure 17 shows the kind of information available from out software toolkit. The top left window shows the main simulation window where you can see how the different parts of the eDNA program is mapped to the eDNA architecture as well as follow package transfers as the propapate through the network in real-time. The bottom window shows a Gantt chart representation of the execution of the eDNA program on the eDNA architecture. The difference colored boxes represents different types of communication, such as start signal, data and handshaking. Finally, the right window shows a task graph representation of the eDNA program.

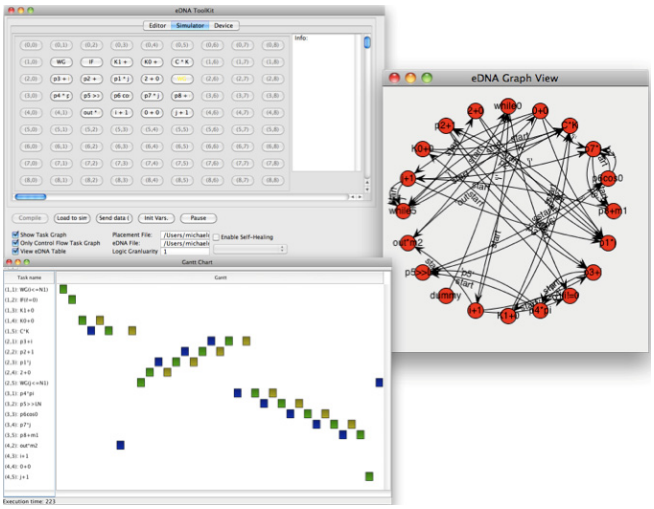

Figure 17 eDNA SW Toolkit

Self-healing example

We will now show an example, which shows how the self-healing works. In this scenario we will inject a fault in eCell at position (1,2) – the initial placement is seen in Figure 18.

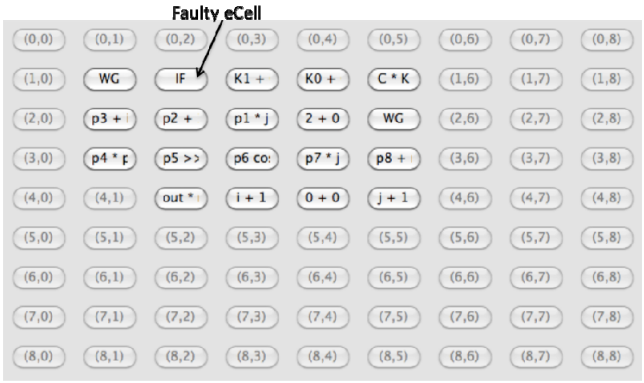

Figure 18 Initial placement of DCT application

The resulting Gantt chart in Figure 19 shows how an example of the execution of the self-healing algorithm described in section 5.

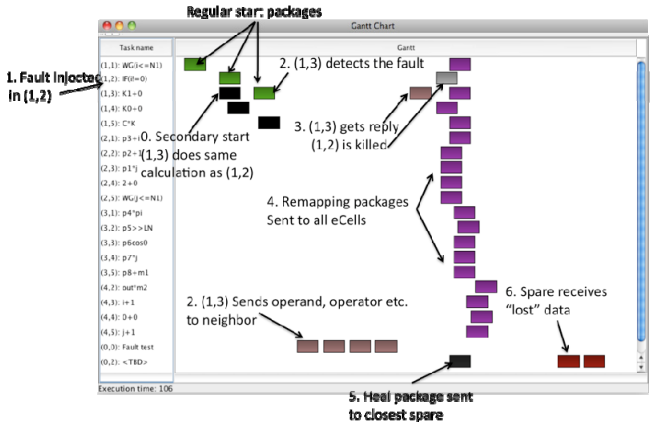

Figure 19 Self-healing example

Figure 20 shows the resulting mapping after the self-healing.

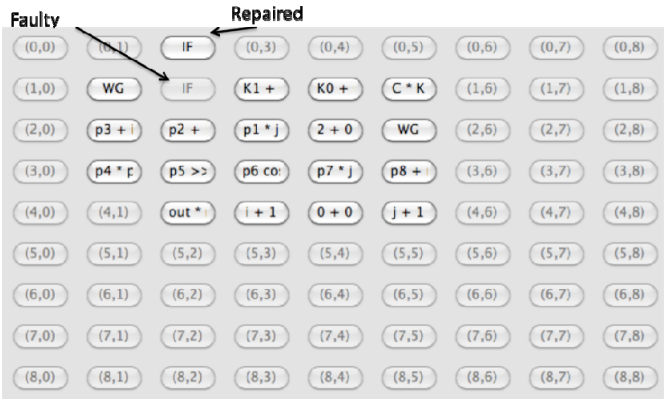

Figure 20 Resulting mapping after self-healing

8. Future work: Advanced features

The future work of this architecture is first and foremost to implement the fault detection and data recovery protocol in the prototype. Once this is done we will be able to test the architecture in a real world fault scenario.

Logical granularity scaling

In previous work [2] we have realized that the logical granularity is of utmost importance to the resulting performance of the application. Fortunately, the eDNA architecture provides a very easy way of scaling the logical granularity up and down.

Currently, the logical granularity at level 1, which means each eCell implement one gene or one arithmetic operation of the eDNA program. With minor modification it is possible to dynamically change the logical granularity of the eDNA architecture by sending a "logical granularity set" package to the eCells telling them how many genes they should implement pr. eCell. Observe that all that is needed to do this is a bigger register file for holding A and B. The eDNA is exactly the same.

This will add a very important feature we can call *dynamical application scaling*. This feature has several important benefits:

1. The programmer is now capable of scaling the area of his application up and down dynamically.

2. The programmer is now capable of scaling the performance of his application up and down dynamically.

3. For an application occupying N eCells, eDNA will now be capable of repairing N-1 additional faults, because when eDNA runs out of spare-eCells it can increase the level of logical granularity hereby making more spare-eCells available.

Currently, we have the dynamical application scaling working in our software model of the eDNA architecture. Figure 21 shows a screenshot of our simulator, which shows what the DCT application would like with a level of logical granularity at 2. Clearly we have now have 9 additional available eCells and the application is now more compact.

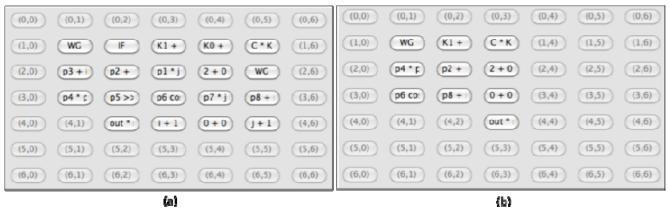

Figure 21 DCT with (a) a level of logical granularity of 1, (b) a level of logical granularity of 2

9. Conclusion

This paper has provided an update on the new developments of the eDNA architecture. We have formalized the model of the eDNA architecture in order to introduce the highly important fault detection and self-healing protocol. In addition, we have described our prototype implementation as well as the porting of it to an embedded system for use with any application. Finally, we have described how to implement the widely used Discrete Cosine Transform on the eDNA architecture as well as demonstrated through an example how the eDNA architecture is capable of self-healing.

REFERENCES

[1] Boesen, M.R., Madsen, J.: eDNA: A bio-inspired reconfigurable hardware cell architecture supporting self-organisation and self-healing. Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware Systems (2009) 147–154.

[2] M. R. Boesen, P. Schleuniger, and J. Madsen. Feasibility study of a self-healing hardware platform. Proceedings of the 2010 Conference on Applied Reconfigurable Computing, 2010.

[3] Mange, D., Sipper, M., Stauffer, A., Tempesti, G.: Toward robust integrated circuits: The embryonics approach. Proceedings of the IEEE 88(4) (2000) 516–543

[4] Stauffer, A., Rossier, J.: Self-testable and self-repairable bio-inspired configurable circuits. 2009 NASA/ESA Conference on Adaptive Hardware Systems (2009) 155–162

[5] Plaks, T., Zhang, X., Dragffy, G., Pipe, A., Gunton, N., Zhu, Q.: A reconfigurable self-healing embryonic cell architecture. International Conference on Engineering of Reconfigurable Systems and Algorithms - ERSA'03 (2003) 134–40

[6] Samie, M., Dragffy, G., Popescu, A., Pipe, T., Melhuish, C.: Prokaryotic bio-inspired model for embryonics. 2009 NASA/ESA Conference on Adaptive Hard- ware Systems (2009) 163–170

[7] I. Page. Constructing hardware-software systems from a single description. Journal of VLSI Signal Processing, (12):87–107, 1996.

[8] Abramovici, M., Strond, C., Hamilton, C., Wijesuriya, S., Verma, V.: Using roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications. Proceedings of IEEE Computer Society International Test Conference (ICSM'99), 973-982, 1999

[9] Xilinx: Microblaze processor reference guide - edk 10.1i. Xilinx User Guide UG081 (v9.0) (2008)

[10] Chapman, K.: Picoblaze 8-bit embedded microcontroller for spartan-3, virtex-ii, and virtex-ii pro fpgas. Xilinx User Guide UG129 (v1.1.2) (2008)

[11] Boesen, M.R., Keymeulen, D., Madsen, J., Lu, T., Chao., T.: Integration of the Reconfigurable Self-Healing eDNA Architecture in an Embedded System and Evaluation of it using a Fourier Transform Spectrometer Instrument Application, To be published in the Proceedings of IEEE Aerospace Conference, 2011.

BIOGRAPHY

Michael Reibel Boesen *is a PhD-student from the Technical University of Denmark (DTU). He earned his Master of Science in Engineering from DTU in 2008 and expects to get his PhD degree in the summer of 2011. He is the co-inventor on the patent-application for the eDNA architecture. His research interests include adaptive and autonomous embedded systems. Michael is the vice-chair of the IEEE Student Branch DTU.*

Jan Madsen *is Professor in computer-based systems at DTU Informatics at the Technical University of Denmark. He is Deputy Head of DTU Informatics and Head of the Section on Embedded Systems Engineering. He is the leader of the Hardware Platforms and Multiprocessor System-on-Chip Cluster within the European Union Network of Excellence on Embedded Systems, ArtistDesign. Jan Madsen is the lead delegate for Denmark in the Governing Board of the ARTEMIS Joint Undertaking, a new pan-European research initiative for public-private partnership in Embedded Systems. He has been Program Chair for DATE (International conference on Design, Automation and Test in Europe) and Program and General Chair for CODES (International conference onHardware/Software Codesign). Jan is the other co-inventor on the patent-application for the eDNA architecture.*

Didier Keymeulen *received the BSEE, MSEE and Ph.D. in Electrical Engineering and Computer Science from the Free University of Brussels, Belgium in 1994. In 1996 he joined the computer science division of the Japanese National Electrotechnical Laboratory as senior researcher. Currently he is principal member of the technical staff of JPL in the Bio-Inspired Technologies Group. At JPL, he is responsible for DoD and NASA applications on evolvable hardware for adaptive computing that leads to the development of fault-tolerant electronics and autonomous and adaptive sensor technology. He participated also as test electronics lead, to Tunable Laser Spectrometer instrument on Mars Science Laboratory. He served as the chair, co-chair, and program-chair of the NASA/ESA Conference on Adaptive Hardware. Didier is a member of the IEEE.*